IN THE SPECIFICATION

Please amend the paragraph 0003 at page 1, line 12 to page 2, line 3, as follows:

Software testing tends to consume a substantial portion of total software development effort, and software maintenance can amount to up to 60% of life cycle costs. A common problem in both software testing and maintenance is deficient documentation, wherein the documentation is often non-existent, inadequate or outdated. These deficiencies can occur for a number of reasons, including inadequate analysis and design effort, requirements changes and changes to code, wherein corresponding updates to the design are not made. Design recovery from object-oriented (OO) programs such as ~~Java~~ JAVA™ by Sun Microsystems and C++ has been proposed, as a means to obtain design information when accurate documentation is not available. Previously, techniques have been proposed for reverse engineering ~~Java~~ JAVA™ and C++ programs to uncover static relationships among objects, such as inheritance, aggregation and association, member function control and data flow relationships, and object state dependent behavior. Reverse engineering is generally the process of analyzing a subject to identify system components and their relationships, and to create representations of a system in another form at a higher level of abstraction.

Please amend the paragraph 0009 at page 3, lines 8-16, as follows:

All reversed engineering environment need tools for extracting the information to be analyzed. Static information includes software artifacts and their relations. In ~~Java~~ JAVA™, for example, such artifacts may be classes, interfaces, methods and variables. The relations might include extension relationships between classes or interfaces, and calls between methods. Dynamic information contains software artifacts as well, and in addition contains sequential event trace information, information about current behavior and code convergence. Static information can be extracted, for example, by using parser-based grammars. For extracting dynamic information, a debugger, profilers, or event recorders can be used.

Please amend the paragraph 0011 at page 4, lines 1-15, as follows:

In order to facilitate understanding, test generation, debugging and documentation, an embodiment of the invention has been developed to provide a reverse engineering technique for design recovery of object interactions from OO programs, such as ~~Java,~~ JAVA™ or C++. Moreover, the embodiment is adapted to represent the artifacts of the program in a very useful form, such as in the form of sequence diagrams. The embodiment of the invention usefully comprises a static analysis technique, wherein multiple passes are used to break the complex OO code and uncover object interactions gradually during successive passes. This enables the embodiment of the invention to support certain powerful features of the OO paradigm, such as inheritance and polymorphism. That is, the proposed method can handle calls to member functions of super-classes as well as calls to polymorphic functions. In addition, the embodiment can handle consecutive calls in the form of ~~o.f(...).g(...).h(...).k(...)~~ O.f(...).g(...).h(...).k(...), where "O" represents an object reference, and "f," "g," "h" and "k" each represent methods, which is very common in ~~Java~~ JAVA™ programming. It is to be understood, however, that embodiments of the invention are not limited to either ~~Java~~ JAVA™ or to representation by means of sequence diagrams.

Please amend the paragraph 0016 at page 5, line 25 to page 6, line 6, as follows:

A very useful embodiment of the invention may be employed to recover, or uncover, object interaction features from the source code of a ~~Java~~ JAVA™ program, although other embodiments may be used in connection with C++ or other programming languages. Generally, respective methods are detected and parsed to gain insight into object interactions. The object interactions are then graphically depicted in the form of a sequence diagram, using a sequence diagram Generator (SDG) constructed in accordance with an embodiment of the invention. The sequence diagram preferably uses a Unified Modeling Language (UML), although other languages can alternatively be used.

Please amend the paragraph 0019 at page 7, lines 3-12, as follows:

Referring to Figure 2, there is shown SDG 18 comprising an embodiment of the invention. SDG 18 includes a Method Information Parser 20, a Method Detail Parser 22, a Diagram Generator 24 and a Drawing Engine 26. Figure 2 further shows a Graphic User Interface (GUI) 28 connected to interact with Method Information Parser 20, Diagram Generator 24 and Drawing Engine 26, and ~~Java~~ JAVA™ (or C++) Source Code 30 is accessed by both Information Parser 20 and Detail Parser 22. The output of Method Information Parser 20 is provided as the input to Method Detail Parser 22, and the output thereof is placed in Method Detail Data Base, or repository, 32. Diagram Generator 24 is coupled to Data Base 32 and to Sequence Diagram Data Base 34, and Drawing Engine 26 is likewise coupled to Data Base 34.

Please amend the paragraph 0021 at page 8, lines 1-6, as follows:

The Sequence Diagram Generator 24 follows the depth first search strategy. That is, the recursive search is stopped when the Sequence Diagram Generator 18 encounters a ~~Java~~ JAVA™ Application Programming Interface (API) in the source code. In the embodiment, ~~Java~~ JAVA™ APIs are treated as black-boxes and not expanded. After all method calls are processed, the entries for the sequence diagram are passed to the Drawing Engine 26 to draw the sequence diagram.

Please amend the paragraph 0026 at page 8, lines 22-25, as follows:

Method Operation Parser 20 can also extract information regarding all the packages or classes that are imported in the ~~Java~~ JAVA™ file being parsed, and this information is likewise stored in the data repository 32 in an ASCII file. This information can be helpful in later stages of parsing, if it is necessary to determine the return type of a method.

Please amend the paragraph 0038 at page 9, line 14 to page 10, line 2, as follows:

Method Detail Parser 22 overcomes several technical difficulties, including a difficulty caused by the common practice of OO programming of making consecutive function calls using a single statement of the form object O.f (...) g(...) h().... For the purpose of sequence diagram generation, it is necessary to determine the classes associated with the objects making the function calls. Thus, in the above example, it is necessary to obtain the class for object, that is the class for the object resulting from the call to object O.f (...) . . . Another difficulty overcome by Detail Parser 22 is associated with the casting operation, which changes the type or class of an object. The casting operation can appear anywhere, including consecutive method calls in a single statement. This poses some difficulty in the static analysis of programs. A further challenge overcome is the handling of complex actual parameters that are passed to a method during a call. For example, an actual parameter may itself be a consecutive function call in a single phrase like object O.f (x.g(...)h(...)...). Finally, Parser 22 is able to differentiate and properly process both application defined functions and ~~Java~~ JAVA™ APIs.

Please amend the paragraph 0044 at page 12, lines 17-21, as follows:

This is the main phase, which finds out the types for all variables and method calls. In this phase, the output provided by the Separating Continuous Method Calls phase is parsed to find the correct variable types (including the phase variables). If a standard ~~Java~~ JAVA™ API is called, then reflection and the imports file generated by Method Information Parser 20 is used to get the return type.

Please amend the paragraph 0045 at page 13, lines 2-10, as follows:

Since ~~Java~~ JAVA™ is a new programming language and still evolving, it is inefficient to keep a large record for all standard ~~Java~~ JAVA™ methods (API's) and use it to distinguish between them and user-defined methods. To distinguish between standard ~~Java~~ JAVA™ APIs

and user-defined methods, a preferred approach is to use the symbol table created by ORD. This table contains all the user-defined classes and file locations for the classes. To determine if a method is a user defined method, the class name of the object containing this method is first retrieved. This class name is then compared with all the class names in the symbol table (this comparison includes package name and class name). If this class name is recorded in the symbol table, then it is a user-defined method; otherwise, it is a standard Java ~~Java~~ JAVA™.

Please amend the paragraph 0046 at page X, lines XX, as follows:

Reference was previously made to inheritance, one of the powerful features of modern object-oriented programming language. Inheritance implies that methods defined for a superclass are automatically defined for its subclasses. That is, a call to a method of an object of a subclass may in fact invoke a method defined in the superclass. The Method Detail Parser 22 utilizes the inheritance relationships captured by ORD. When encountering a method call, it first tracks the inheritance path until it reaches a parent class in which the method is defined. If no such class is found, then the parser uses the import clauses and reflection to trace the ~~Java~~ JAVA™ APIs, to identify the class that defines the method. Another powerful feature in OO programming is polymorphism. Polymorphism allows programmers to write multiple methods with the same name but different parameter types or number of parameters. In the approach of the invention, polymorphism is solved by using reflection and a symbol table generated by ORD. The symbol table entry contains the following information:

Please amend the paragraph 0048 at page 13, line 26 to page 14, line 5, as follows:

Reflection is used to retrieve class-related information such as class attributes and methods. Since a method can be either a standard ~~Java~~ JAVA™ API or a user-defined method, it is necessary to match the correct method separately. If it is decided that the method call is a standard ~~Java~~ JAVA™ API, reflection is used to get the return type of the standard ~~Java~~ JAVA™ API. The type is then used to update the variable table in case it is used as one of the parameters

in another method call. On the other hand, if this method call is a user-defined method call, the data in the ORD symbol table is searched for the user-defined method to match with the correct version of the method call. As in retrieving the return type in standard ~~Java~~ JAVA™ API, the return object type is updated in the variable table as well.